The Patent Office
Concept House
Cardiff Road
Newport
South Wales
NP10 8QQ

I, the undersigned, being an officer duly authorised in accordance with Section 74(1) and (4) of the Deregulation & Contracting Out Act 1994, to sign and issue certificates on behalf of the Comptroller-General, hereby certify that annexed hereto is a true copy of the documents as originally filed in connection with the patent application identified therein.

In accordance with the Patents (Companies Re-registration) Rules 1982, if a company named in this certificate and any accompanying documents has re-registered under the Companies Act 1980 with the same name as that with which it was registered immediately before re-registration save for the substitution as, or inclusion as, the last part of the name of the words "public limited company" or their equivalents in Welsh, references to the name of the company in this certificate and any accompanying documents shall be treated as references to the name with which it is so re-registered.

In accordance with the rules, the words "public limited company" may be replaced by p.l.c., plc, P.L.C. or PLC.

Re-registration under the Companies Act does not constitute a new legal entity but merely subjects the company to certain additional company law rules.

**CERTIFIED COPY OF PRIORITY DOCUMENT**

Signed

Dated 17 March 2004

# Patents Form 1/77

THE PATENT OFFICE
J
- 4 JUL 2003
NEWPORT

The Patent Office
Cardiff Road
Newport
NP9 1RH

## Request for grant of a patent

07JUL03 E820681-1 D00201
P01/7700 0.00-0315844.1

| | | |
|---|---|---|
| 1. Your Reference | **IMR/CEE/Y1382** | |
| 2. Application number | 04 JUL 2003 | 0315844.1 |
| 3. Full name, address and postcode of the or each Applicant<br><br>Country/state of incorporation (if applicable) | **Transitive Limited**<br>**5th Floor Alder Castle**<br>**10 Noble Street**<br>**London**<br>**EC2V 7QJ**<br><br>**Incorporated in: United Kingdom** | |
| 4. Title of the invention | **Method and Apparatus for Performing Adjustable Precision Exception Handling** | |
| 5. Name of agent<br><br>Address for service in the UK to which all correspondence should be sent<br><br>Patents ADP number | **APPLEYARD LEES**<br><br>**15 CLARE ROAD**<br>**HALIFAX**<br>**HX1 2HY**<br><br>**190001** | |
| 6. Priority claimed to: | Country | Application number / Date of filing |
| 7. Divisional status claimed from: | Number of parent application<br>- | Date of filing<br>- |
| 8. Is a statement of inventorship and of right to grant a patent required in support of this application? | **YES** | |

C8664211001

9. Enter the number of sheets for any of the following items you are filing with this form. Do not count copies of the same document

| | |
|---|---|
| Continuation sheets of this form | - |
| Description | 23 |
| Claim(s) | 2 |
| Abstract | 1 |
| Drawing(s) | 3 |

10. If you are also filing any of the following, state how many against each item

| | |
|---|---|
| Priority documents | - |
| Translation of priority documents | - |
| Statement of inventorship and right to grant a patent (PF 7/77) | - |
| Request for a preliminary examination and search (PF 9/77) | - |
| Request for substantive examination (PF 10/77) | - |
| Any other documents (please specify) | - |

11. We request the grant of a patent on the basis of this application.

Signature — Date

**APPLEYARD LEES** — **01 July 2003**

12. Contact — **Ian Robinson- 01422 330110**

# METHOD AND APPARATUS FOR PERFORMING ADJUSTABLE PRECISION EXCEPTION HANDLING

The subject invention relates generally to the field of computers and computer software and, more particularly, to program code conversion methods and apparatus useful, for example, in code translators, emulators and accelerators which encounter exception signals.

An exception is an error condition that changes the normal flow of control in a program. An exception may be generated ("raised") by hardware or software. Hardware exceptions include such signals as resets, interrupts or signals from a memory management unit. Exceptions may be generated by an arithmetic logic unit or floating-point unit for numerical errors such as divide by zero, for overflow or underflow, or for instruction decoding errors such as privileged, reserved, trap or undefined instructions. Software exceptions are varied respectively across various software programs and could be applied to any kind of error checking which alters the normal behaviour of the program. An exception handler is special code which is called upon when an exception occurs during the execution of a program. If the program does not provide a handler for a given exception, a default system exception handler will be called, usually resulting in abortion of the program being run and an error indication being returned.

Exception signals are a common mechanism for raising exceptions on many operating systems. The POSIX standard, which is adhered to by many operating systems,

particularly Unix-like systems, specifies how this mechanism should behave so that exception signals are broadly similar across many systems. The most common events that trigger exceptions are when a process implemented by a program tries to (i) access an unmapped memory region or (ii) manipulate a memory region for which it does not have the correct permissions. Other common events that trigger exception signals are (iii) a signal is sent from another process, (iv) the process tries to execute an instruction that it does not have the privilege level to execute, or (v) an I/O event in the hardware.

Due to the interruption of the program being executed, the delivery of an exception signal by the operating system normally includes a captured state of the subject processor when the exception occurred. This state can be very difficult to determine and costly to generate. In order to avoid these costs, it is generally preferable to avoid intentionally issuing exceptions unless there are no better alternatives.

Some representative exception signals issued by operating systems to define certain events are described in Table 1.

| Signal | Description |
|---|---|
| SIGHUP | "Hangup" – commonly used to indicate to a process that its configuration has changed, and that it should re-read its config file. |
| SIGINT | "Interrupt" – usually means Ctrl-C has been pressed by the user. |
| SIGILL | "Illegal Instruction" – the processor generates this when an invalid instruction opcode is encountered. |
| SIGTRAP | "Breakpoint" – often used by debuggers. |

| Signal | Description |
|---|---|
| SIGBUS | "Bus Error" – usually generated by the processor to indicate an invalid memory access. This is usually an access to an unallocated or unaligned memory address. |
| SIGSEGV | "Segmentation Violation" – generated by the processor when a user process has tried to do something not permissible in user mode. For example, trying to execute a privileged instruction, or trying to write to part of the kernel memory would both raise this signal. |
| SIGALRM | "Alarm Clock" – a process can make the alarm() system call, which requests the delivery of this signal *n* seconds later. |
| SIGTERM | "Terminate" – polite request for a program to think about exiting, if it's not too inconvenient. |
| SIGQUIT | "Quit" – Firm request for a program to exit, now please! |
| SIGKILL | "Die" – immediately terminates the process. This signal cannot be intercepted by a signal handler. |

**Table 1: Exception Signals**

Exception signals can come from two sources: (1) directly from a subject program or (2) from the operating system or another process. Some exception signals are generated as a direct result of an instruction executed by the subject program. For example, if a subject program executes an illegal opcode then SIGILL is raised. Similarly, if the subject program attempts an illegal memory access then SIGSEGV is raised. These are referred to in-band signals. Exception signals can also be generated externally, either by the operating system or by another process. SIGHUP and SIGALRM are examples of these. These externally generated exception signals are called out-of-band signals.

From a subject program's point of view, an exception signal can occur at any time. When a signal occurs, the operating system interrupts the execution of the signaled program and invokes a signal handler function. The operating system maintains a process-specific function table which maps each signal to a particular signal

handler. The operating system also defines default signal handlers for all exceptions. The default signal handlers either take predefined actions or simply ignore the signal.

For example, in Unix, a program can override a default signal handler by invoking the sigaction() system call. Sigaction() allows the program to specify what action the operating system should take when a particular exception signal is received. The action can be: (1) ignore the exception signal; (2) call the default signal handler; or (3) call a specialized signal handler function, whose address is provided by the program. Other options that can be specified when making the sigaction() call include which other signals are blocked during execution of a signal handler, in much the same way as a CPU can mask certain interrupts.

A Unix signal handler can be provided with one of two prototypes. The first signal handler prototype is "void sigHandler(int sigNum)." The first argument is the number of the exception signal, so that one function can be registered to handle multiple signals. A program can request that more information be provided to the signal handler by calling sigaction() with the SA_SIGINFO flag. In this case, the Unix signal handler prototype becomes "void sigHandler(int sigNum, siginfo_t sigInfo, void *context)."

The second parameter ("siginfo") is a structure which contains information about the signal, including some indication of what caused the signal and where it came from. For example, in the case of a SIGILL signal the

siginfo structure contains the address of the illegal instruction. This data can be essential to allow the process to handle the signal properly. The third parameter ("context") provides access to the processor state (including all subject registers) at the time the signal was raised. Again, this data can be essential to allow correct handling of a signal. The signal handler is allowed to modify this context; when execution is resumed, the subject registers are then restored to the values of the modified context.

In both embedded and non-embedded CPU's, one finds predominant Instruction Set Architectures (ISAs) for which large bodies of software exist that could be "accelerated" for performance, or "translated" to a myriad of capable processors that could present better cost/performance benefits, provided that they could transparently access the relevant software. One also finds dominant CPU architectures that are locked in time to their ISA, and cannot evolve in performance or market reach. Such architectures would benefit from "Synthetic CPU" co-architecture.

Program code conversion methods and apparatus facilitate such acceleration, translation and co-architecture capabilities and are addressed, for example, in the co-pending patent application, UK Application No. 03 09056 0, entitled Block Translation Optimizations for Program Code Conversion, and filed on April 22, 2003, the disclosure of which is hereby incorporated by reference.

According to the present invention there is provided an apparatus and method as set forth in the appended

claims. Preferred features of the invention will be apparent from the dependent claims, and the description which follows.

The following is a summary of various aspects and advantages realizable according to various embodiments according to the invention. It is provided as an introduction to assist those skilled in the art to more rapidly assimilate the detailed design discussion that ensues and does not and is not intended in any way to limit the scope of the claims that are appended hereto.

In particular, the inventors have developed an optimization technique directed at expediting program code conversion, particularly useful in connection with a run-time translator which employs translation of subject program code into target code. An adjustable precision exception handling technique is providing for handling exceptions encountered during execution of translated subject code at varying levels of precision, depending upon the particular type of exception encountered. As an exception signal is detected by the translator, the state of the subject processor is captured at a level of precision determined to be sufficient for the detected exception and the corresponding signal handling behavior of the subject program.

The accompanying drawings, which are incorporated in and constitute a part of the specification, illustrate presently preferred implementations and are described as follows:

Fig. 1 is a schematic diagram illustrating exception signal handling processes in accordance with an illustrative embodiment of the invention;

Fig. 2 is a schematic diagram illustrating exception signal handling processes in accordance with an illustrative embodiment of the invention; and

Fig. 3 is a block diagram of apparatus wherein embodiments of the invention find application.

Illustrative apparatus for implementing various novel features discussed below is shown in Figure 3. Figure 1 illustrates a target processor 13 including target registers 15 together with memory 18 storing a number of software components 17, 19, 20, 21 and 22. The software components include subject code 17 to be translated, an operating system 20, the translator code 19, the translated code 21, and a variable precision exception handling mechanism 22. The translator code 19 may function, for example, as an emulator translating subject code of one ISA into translated code of another ISA or as an accelerator for translating subject code into translated code, each of the same ISA.

The translator 19, i.e., the compiled version of the source code implementing the translator, and the translated code 21, i.e., the translation of the subject code 17 produced by the translator 19, run in conjunction with the operating system 20 such as, for example, UNIX running on the target processor 13, typically a microprocessor or other suitable computer. It will be appreciated that the structure illustrated in Figure 3 is

exemplary only and that, for example, software, methods and processes according to the invention may be implemented in code residing within or beneath an operating system. The subject code, translator code, operating system, and storage mechanisms may be any of a wide variety of types, as known to those skilled in the art.

In apparatus according to Fig. 3, program code conversion is preferably performed dynamically, at run-time, while the translated code 21 is running. The translator 19 runs inline with the translated program 21. If a exception signal handler is registered by the subject code 17, the translator 19 must translate and execute the subject code of the signal handler on receipt of the corresponding exception signal. The translator 19 must correctly emulate the subject operating system semantics by keeping track of which respective portions of the subject code 17 to execute upon the receipt of respective exception signals. In addition, the same exception signal may have different signal numbers on different architectures, so emulation may include translation of signal numbers. It is possible for the translator 19 to detect a subject program exception during decoding of the subject code 17, where the translator 19 will then simply invoke the subject signal handler at the appropriate time rather than raising an actual exception signal.

The translator 19 must populate the siginfo data structure if it is used by the subject signal handler. Likewise, the translator 19 must populate the sigcontext data structure if it is used by the subject signal handler. The sigcontext structure contains a snapshot of

the subject processor state at the time of the exception signal. Since some signals, such as SIGSEGV, can be raised by almost any instruction, populating the sigcontext structure requires that the translator 19 be able to correctly regenerate the entire subject processor state from an arbitrary point within a block of subject code 17. The ability to recreate the entire subject processor state is referred to as precise exception handling. Precise exception handling is very difficult for a dynamic translator to support without significant performance losses.

In cases where a subject code 17 registers a signal handler, the translator 19 must emulate the semantics of that signal handler. In most cases, the subject code's signal handler is not executable on the target architecture so the target signals must be handled by proxy. The translator 19 registers a proxy signal handler in the target program 21, which intercepts signals from the target operating system 20, constructs the appropriate subject context, and invokes the (translated) subject signal handler. Alternatively, if the translator 19 detects at decode-time that a particular subject code instruction will raise an exception signal, the translator 19 plants target code which constructs the subject context and invokes the subject signal handler directly, without raising an actual signal on the target operating system 20. This direct invocation mechanism is also referred to as a proxy signal handler.

When an exception occurs in the translated subject program, the proxy signal handler is invoked. If the exception occurs as an exception signal issued by the

target operating system 20, the target operating system 20 invokes the proxy signal handler. Alternatively, if the exception is detected by the translator 19 during decoding of the subject code 17, the target code 21 invokes the proxy signal handler. The proxy signal handler translates the subject program's signal handler, recreates the subject state, and then passes the subject state to the translated signal handler. The subject state is recreated using several sources, including the translator 19's representation of the subject processor state and the target processor state passed to the proxy handler.

Figure 1 illustrates the control flow of signal handling in a translated program. The left side of Figure 1 shows signal handling which occurs when a program is running on its subject architecture (i.e., not within a translator). The right side of Figure 1 shows signal handling for a translated program using the target operating system signal mechanism of a present embodiment. In the subject architecture signal handling scenario on the left side of Figure 1, signal handling begins when the subject program 101 raises an exception signal 111, which transfers control to the operating system 103. The operating system 103 constructs a context structure 113 which reflects the state of the subject processor when the signal 111 occurred. The context structure is then passed to the signal handler 105 that was registered for the particular signal encountered 111. When the signal handler 105 completes its operation, the context structure 113' is returned to the operating system 103. The operating system 103 then restores the processor state from the context 113' and returns control to the program 101. Any changes to the context structure 113 (e.g., to a

register value) made by the signal handler 105 will be reflected in the processor state when the program 101 resumes execution. In Figure1, the apostrophe at the end of the context sructure 113' indicates that the contents of the context structure may have been modified.

The right side of Figure 1 illustrates the control flow of signal handling in a translated program 21 executed by the target processor 13, using the signal mechanism of the target operating system 20. To emulate signal handling, the translator 19 first registers a proxy signal handler function 125 for the translated program 121. When a signal 131 occurs in the translated program 121, the target operating system 20 invokes the proxy signal handler 125 and passes it the target context 133. The proxy signal handler 125 uses the target context 133 and a subject register bank 141 to construct the subject state 135. The translator 19 then identifies, translates, and executes the corresponding subject program signal handler 127 and passes it the reconstructed subject state 135. When the subject signal handler 127 completes it operation, the subject context 135' is returned to the proxy signal handler 125. When the proxy signal handler completes it operation, it returns the target context 133' to the target operating system 20. The target operating system 20 then restores the target processor state using the target context 133' and resumes execution of the translated subject code 121.

The subject register bank 141, otherwise referred to as a global register store, is a memory region that is a repository for abstract registers, each of which corresponds to and emulates the value of a particular

subject register or other architectural feature. During the execution of translated code 21, abstract register values are stored alternatively in the subject register bank 141 or target registers 15. During the execution of translated code 21, abstract registers are temporarily held in target registers 15 so that they may participate in instructions. Subject register values are saved in the subject register bank 141 when they are not being held in target registers. For example, if the register allocation for a particular block of subject code requires a subject register value to be spilled (i.e., saved to memory in order to free a target register), the value is spilled to a reserved location within the subject register bank that corresponds to that particular subject register. Likewise, all subject register values are saved back to the subject register bank 141 at the end of each block of subject code, as target register values may be overwritten between successive blocks. The various features associated with creating and using a subject register bank 141 or global register store are described in detail in co-pending patent application, U.S. Patent Application Serial No. 10/439,966, filed on May 16, 2003 and assigned to the same assignee as the present application. The details of the global register store described in Serial No. 10/439,966 are hereby incorporated by reference into the present application.

In cases where the subject signal handler 127 modifies the subject context 135, the proxy signal handler saves the modified values to the subject register bank 141. In Figure1, the apostrophe at the end of subject context 135' indicates that the contents may have been modified. If, at the time the signal was raised, any of those subject

register values were live in target registers, the proxy signal handler 125 also updates the corresponding entries in the target context 133. In Figure 1, the apostrophe at the end of target context 133' indicates that the contents may have been modified.

As discussed in greater detail hereafter, depending on the level of precision required by the subject signal handler 105, the translator 19 may plant target code immediately preceding the instruction that raises the exception to rectify and spill all subject register values to the subject register bank 141. This guarantees that the values retrieved from the subject register bank 141 by the proxy signal handler 125 are accurate, and the subject context 135 passed to the translated subject signal handler 127 is accurate.

Figure 2 illustrates the control flow of signal handling for a translated program 21, using direct invocation of the proxy signal handler by target code, and in which the subject signal handler does not modify the subject context 113. The left side of Figure 2 shows signal handling when a program is running on its subject architecture. The right side of Figure 2 shows signal handling for a translated program using the direct invocation mechanism. In the subject architecture signal handling scenario on the left side of Figure 2, signal handling begins when the subject program 101 raises a signal 111, which transfers control to the operating system 103. The operating system 103 constructs a context structure 113 which reflects the state of the processor when the signal 111 occurred. The context structure is then passed to the signal handler 105 that was registered

for the particular signal encountered 111. When the signal handler 105 completes it operation, it returns control to the operating system 103. The operating system 103 then returns control to the program 101.

The right side of Figure 2 illustrates signal handling in a translated program using the direct invocation mechanism. At the point in the translated program 121 where the signal would have occurred, the target code 21 constructs a target context 133 and invokes the proxy signal handler 125, passing it the target context 133. The proxy signal handler 125 uses the target context 133 and the subject register bank 141 to construct the subject state 135 as a subject context structure. The translator 19 then identifies, translates, and executes the corresponding subject program signal handler 127 and passes it the reconstructed subject state 135. When the subject signal handler 127 completes, it returns control to the proxy signal handler 125. When the proxy signal handler completes, it returns control to the translated subject code 121, which resumes execution.

Recreating the subject processor state can be both difficult and expensive. First, there is an actual cost associated with calculating and collecting the subject state. For example, translator optimizations such as lazy evaluation may postpone the calculation of subject register values by storing the underlying data necessary to calculate those values. Recreating the subject state in response to a signal requires those values to be rectified (i.e., calculated) immediately. Even if the subject registers have been calculated previously, they

must be retrieved from memory, such as from a subject register bank.

Second, there is an opportunity cost associated with the capability of calculating the subject state at any point in the subject program. Many key optimizations in a dynamic binary translator involve departures from a strict model of binary compatibility. Binary compatibility means that the translator can recreate the exact state of the subject architecture. A strict model is one in which the subject state can be recreated at any point in the translated program (i.e., at any subject instruction). In order to preserve the information necessary to recreate the subject state at any point in execution, the translator must forego significant optimizations. When those optimizations are in use, the translator has no way to recreate the subject context accurately. Thus, the real cost of exceptions is not generating the state when the exception occurs but being able to generate the state at all.

The translator 19 emulates exceptions using a variable precision exception handling mechanism 22, whereby the subject context is reconstructed at different levels of detail for different exceptions, depending on the requirements of the particular exception. The translator 19 detects subject exceptions at decode-time and when encountered during translation, and determines the precision required for the subject context for a particular exception. By comparison, a naïve translator is inefficient and would implement the most conservative solution to allow for the possibility that a complete and precise subject context might be needed at any point.

In one embodiment, the variable precision exception handling mechanism 22 provides exception handling at four levels of subject context precision: (0) no state; (1) the last known consistent stack frame; (2) precise program counter; and (3) full rectified subject registers. The cost of rectifying all subject registers is very high and is therefore avoided if at all possible. Therefore, the translator 19 handles each exception at the lowest level of precision necessary. Generally, the translator 19 can determine at decode-time what level of precision is required. While this embodiment of the variable precision exception handling mechanism 22 describes four levels of subject context precision, it is possible for the variable precision exception handling mechanism 22 to select any level of precision from any number of possible levels of precision. Furthermore, the particular components of the subject context being captured for each level of precision can also be variably selected.

Table 2 illustrates the actions performed by different translator components for each exception handling precision level for this particular embodiment employing the following four selected levels of precision, which are detailed below.

| | Level 0 | Level 1 (Default) | Level 2 | Level 3 |
|---|---|---|---|---|
| Decoding | - | - | **save PC** | **save PC** |
| | - | - | - | **rectify/spill (IR)** |
| Target Code | - | - | - | **rectify subj regs** |
| | - | - | - | **spill subj regs** |
| | exception | exception | exception | exception |
| Proxy Signal Handler | - | load PC (vague) | **load PC (precise)** | **load PC (precise)** |
| | - | load stack frame (vague) | load stack frame (vague) | **load subj regs (precise)** |
| | translate/invoke subject handler | translate/invoke subject handler | translate/invoke subject handler | translate/invoke subject handler |
| | - | - | - | **save subj regs** |
| | - | - | - | **modify target regs** |

**Table 2: Variable Precision Exception Handling**

## Level Zero: No State

In some cases, the translator 19 determines that no subject state is necessary to handle an exception. In these cases, the subject context passed to the translated signal handler need not contain any accurate processor state data. For example, in the Linux operating system, signal number 31 is reserved for the "Pthread" threading library. This signal is sent from a newly created child thread to its parent to signify successful creation. The implementation of the pthread library indicates that the handler requires no state whatsoever, rather the act of delivering of the exception itself is the only data required.

**Level One: Last Known Stack Frame**

Another level of precision is to provide the last known stack frame, meaning the last known consistent values of the stack pointer, base pointer, and program counter registers. In most situations, it is most efficient to make the last known stack frame the default level of precision. The remaining, undefined values in the subject context are filled in with a special value, such as "0xdeadbeef," for debugging purposes.

These values are imprecise in that they demonstrate the last known consistent state of the subject program and may not precisely reflect the current state. In one embodiment of the translator 19, the last known consistent state corresponds to the last basic block boundary, as generally all subject registers are rectified and saved to the subject register bank between basic blocks. The "last known stack frame" level of precision requires no rectification of subject register values.

Regardless of the precision level used, if the signal is the result of a memory access, the translator 19 will also fill in the corresponding subject address that caused the exception. This value is derived from the target state passed to the proxy handler, or encoded in the target code which invokes the subject signal handler. Depending on the memory models of the subject code 17 and target code 21, the translator 19 may need to demangle the target address of the memory access to obtain the corresponding subject address.

**Level Two: Precise Program Counter**

In some cases, the translator 19 determines at decode-time that (a) a particular subject instruction will trigger a signal and (b) the subject signal handler will require a precise program counter value. For example, on the x86 processor, the IRET instruction can only be executed at certain privilege levels; if the translator encounters this instruction at a time when the subject privilege level is too low, proper emulation requires that the corresponding subject signal handler be invoked, and that the signal handler be passed a precise value for the subject program counter.

The translator 19 emulates such instructions by intentionally raising the correct exception or by invoking the subject signal handler directly at the appropriate location. During translation, the translator records the subject address of the instruction that will cause the exception in a temporary abstract register. The translator also sets a flag to indicate the exception handling precision level, so that the proxy signal handler will know how much context to construct when it is subsequently invoked. The "precise program counter" flag tells the proxy signal handler to retrieve the precise program counter value. The translator then plants target code to raise the appropriate exception (normally SIGILL on Unix systems including Linux) or invoke the subject signal handler directly. When the proxy signal handler is invoked, it will note that the "precise program counter" flag is set and retrieve the program counter value from its stored location.

**Level Three: Precise Subject Registers**

The highest precision level of this embodiment is used where the translator 19 detects at decode time that an instruction will cause an exception and will require context beyond the program counter. In this case, the translator 19 records the precise program counter value of the excepted instruction and generates target code to rectify all subject registers. This allows the translator 19 to generate the full subject context when the exception is raised.

In some cases, the translator 19 determines at decode-time that a particular instruction in the subject code 17 will cause an exception and that the exception handling will require some state beyond the program counter. In this case, at decode-time the translator 19 both records the subject address of the instruction and forces all subject register values to be rectified prior to the excepted instruction. In addition, the translator 19 marks the block containing the exception such that, during code generation, certain optimizations are not applied. Some optimizations involving code motion can cause the subject register values to be temporarily (i.e., at certain points in the target code) inconsistent with the subject state even if rectified; these optimizations are turned off for blocks that contain exceptions, to guarantee the accuracy of the subject state passed to the subject signal handler.

For example, the x86 INB instruction is used by some subject programs to emulate hardware accesses. The signal handler in this case requires precise values for all

subject registers (i.e., the full context). When an x86 INB instruction is encountered during translation (decoding), the translator inserts target code (or IR, which is later emitted as target code) to rectify all lazy or pending subject register values and then spill them to the subject register bank. The translated block will thus comprise rectification code, spill code, and an exception-raising target instruction. During the subsequent execution of the translated block, the subject state is rectified and spilled before the exception is raised, such that the values in the subject register bank are consistent and accurate when the proxy signal handler is invoked.

The proxy signal handler 125 constructs the subject context 135 using the rectified values in the subject register bank 141, and constructs the siginfo structure from the stored address of the instruction that raised the signal. The proxy signal handler 125 then translates the subject signal handler 127 and invokes it with the siginfo and context 135 structures. When the subject signal handler 127 finishes execution, it returns control to the proxy signal handler 125. When the proxy signal handler 125 finishes execution, control returns to the translated program 121.

In rare cases, the subject signal handler 127 modifies some of the subject registers in the context structure 135. In this case, the proxy signal handler 125 copies those modified values back into the subject register bank 141 prior to returning control. This ensures that any modified subject register values are propagated into the translated program 121.

Although a few preferred embodiments have been shown and described, it will be appreciated by those skilled in the art that various changes and modifications might be made without departing from the scope of the invention, as defined in the appended claims.

Attention is directed to all papers and documents which are filed concurrently with or previous to this specification in connection with this application and which are open to public inspection with this specification, and the contents of all such papers and documents are incorporated herein by reference.

All of the features disclosed in this specification (including any accompanying claims, abstract and drawings), and/or all of the steps of any method or process so disclosed, may be combined in any combination, except combinations where at least some of such features and/or steps are mutually exclusive.

Each feature disclosed in this specification (including any accompanying claims, abstract and drawings) may be replaced by alternative features serving the same, equivalent or similar purpose, unless expressly stated otherwise. Thus, unless expressly stated otherwise, each feature disclosed is one example only of a generic series of equivalent or similar features.

The invention is not restricted to the details of the foregoing embodiment(s). The invention extends to any novel one, or any novel combination, of the features disclosed in this specification (including any

accompanying claims, abstract and drawings), or to any novel one, or any novel combination, of the steps of any method or process so disclosed.

**Claims**

1. A method of handling exceptions encountered during the translation of program code, comprising:

detecting the occurrence of an exception during translation of the program code;

determining a level of subject context precision required for the particular exception detected from a plurality of possible levels of precision; and

invoking a signal handler to handle the detected exception.

2. The method of claim 1, wherein the exception occurrence detecting step detects when an exception signal occurs in the program code being translated.

3. The method of claim 2, wherein the target code generated by the translation invokes a proxy signal handler to handle the detected exception.

4. The method of any preceding claim, wherein the exception occurrence detecting step detects when an exception signal is received from a target operating system.

5. The method of claim 4, wherein the target operating system invokes a proxy signal handler to handle the detected exception.

6. The method of any preceding claim, wherein the default level of subject context precision is a last known stack frame.

7. The method of claim 6, wherein the last known stack frame includes the last known stack pointer values, base pointer values, and program counter registers.

8. The method of claim 7, wherein said default level of subject context precision requires no rectification of subject register values.

9. The method of any preceding claim, wherein one of said possible levels of subject context precision is no subject state.

10. The method of any preceding claim, wherein one of said possible levels of subject context precision is a precise program counter state including a precise program counter value.

11. The method of any preceding claim, wherein one of said possible levels of subject context precision is a precise subject register state including rectified subject registers and a precise program counter value.

## ABSTRACT

# METHOD AND APPARATUS FOR PERFORMING ADJUSTABLE PRECISION EXCEPTION HANDLING

An adjustable precision exception handling technique is providing for handling exceptions encountered during translation of subject code to target code at varying levels of precision, depending upon the particular type of exception encountered. As an exception signal is detected by the translator, the state of the subject processor is captured at a sufficient precision determined to be appropriate for the detected exception.

[Figure 1]

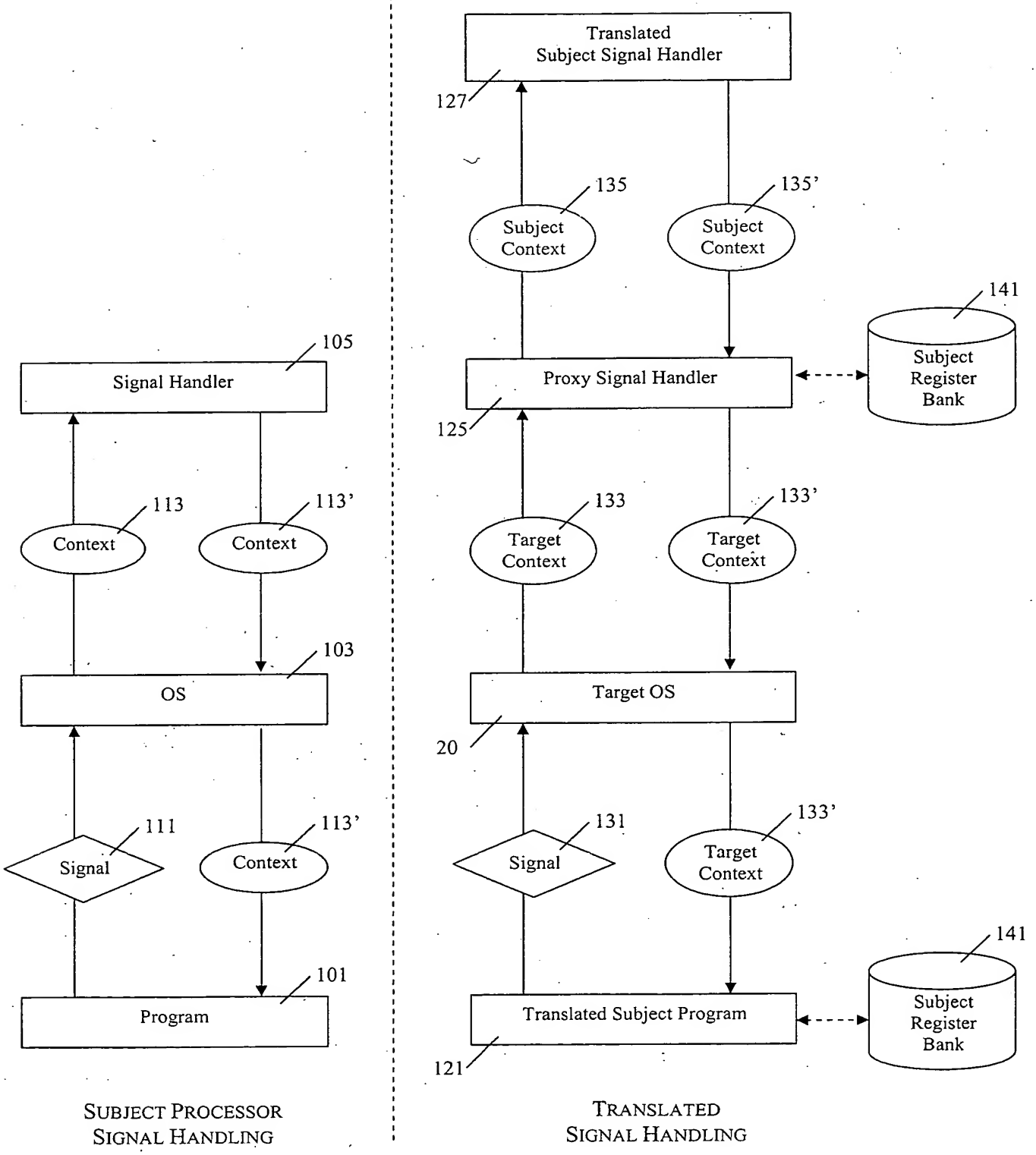
Translated
Subject Signal Handler
127
135
135'
Subject
Context
Subject
Context
141
Subject
Register
Bank
Proxy Signal Handler
125
105
Signal Handler
133
133'
Target
Context
Target
Context
113
113'
Context
Context
103
OS
Target OS
20
111
131
133'
Signal
Context
Signal
Target
Context
141
101
Program
Translated Subject Program
Subject
Register
Bank
121
SUBJECT PROCESSOR
SIGNAL HANDLING
TRANSLATED
SIGNAL HANDLING

FIG. 1

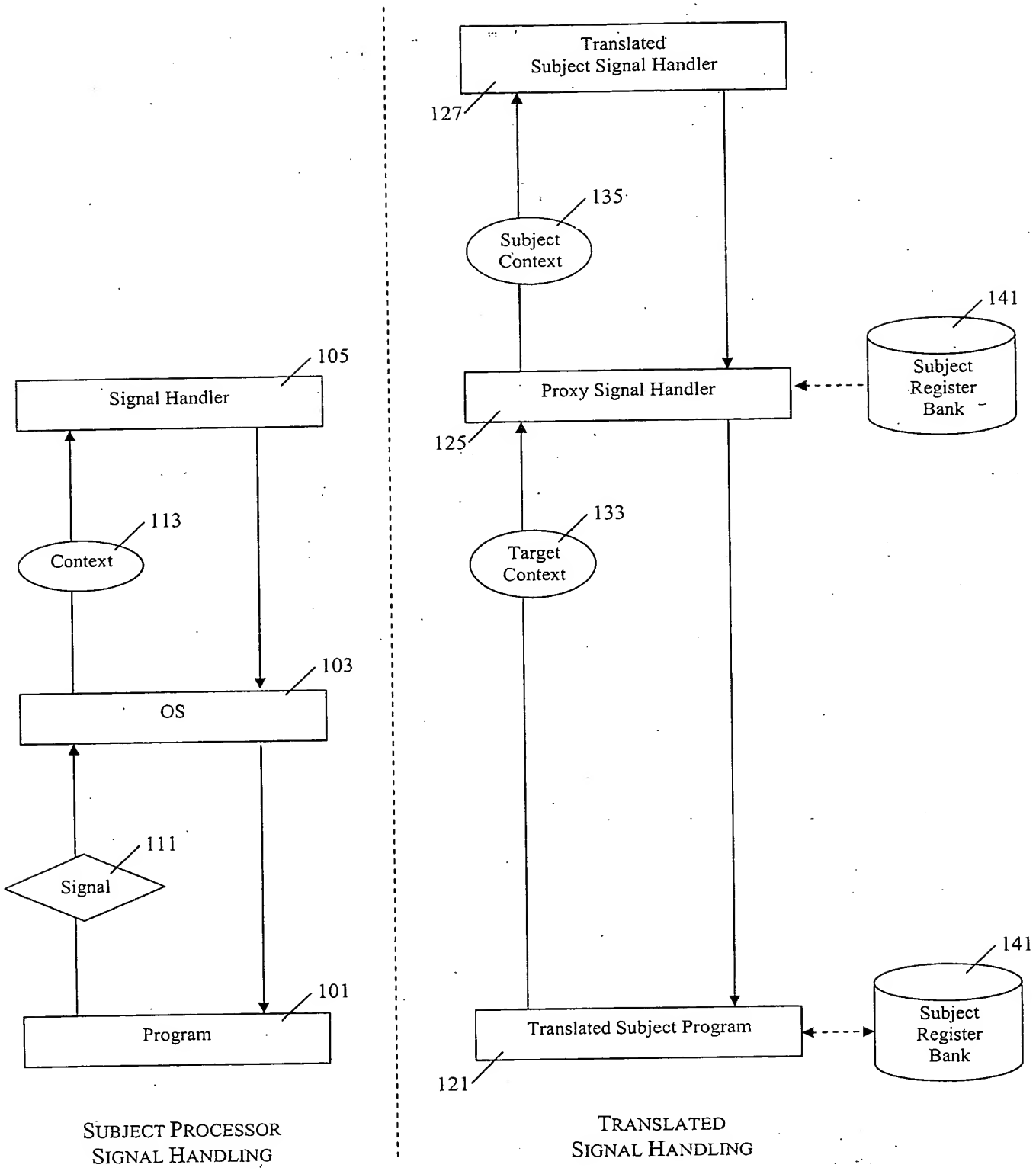
Translated
Subject Signal Handler
127
135
Subject
Context
141
Subject
Register
Bank
105
Signal Handler
Proxy Signal Handler
125
113
Context
133
Target
Context
103
OS
111
Signal
141
101
Program
Translated Subject Program
121
Subject
Register
Bank
SUBJECT PROCESSOR
SIGNAL HANDLING
TRANSLATED
SIGNAL HANDLING

FIG. 2

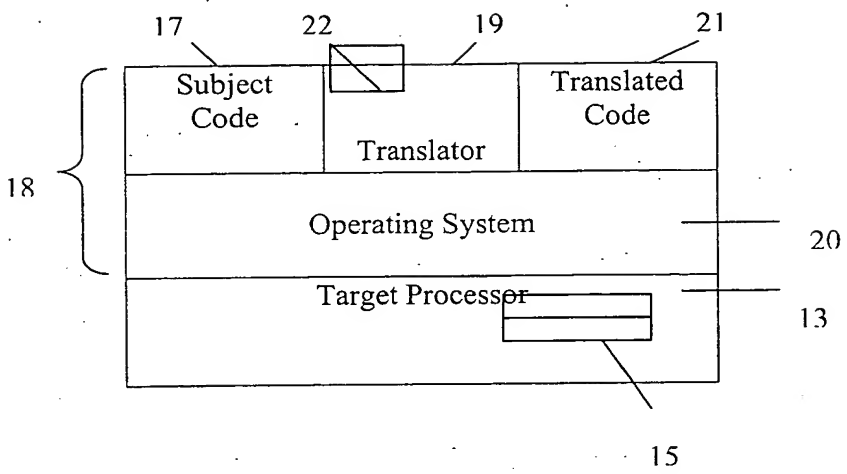
17
22
19
21
Subject Code
Translator
Translated Code
18
Operating System
20
Target Processor
13
15

**FIG. 3**